OBJECT-ORIENTED PROGRAMMING WITH PYTHON

AIOTLAB, September 2023

Table of Contents

Table of Contents

Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Classes and Objects

- Create a Class:
  - To create a class, use the keyword **class**

```
class MyClass:
    x = 5

print(MyClass)

Output: <class '__main__.MyClass'>
```

- Create an Object:
  - After a class is created, it can be used to create objects

```
class MyClass:
    x = 5

p1 = MyClass()
print(p1.x)

Output: 5
```

Classes and Objects

- The **__init__()** Function
  - All classes have a function called __init__()
  - This function is always executed when the class is being initiated
  - In other words, it is called automatically every time the class is being used to create a new object
  - It is similar to **Constructor** in C++ and Java
  - Usage:
    - Assign values to object properties
    - Declare other operations that are necessary to do when the object is being created

Classes and Objects

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person('John', 36)

print(p1.name)
print(p1.age)

Output:
John
36
```

Classes and Objects

- The **__str__()** Function
  - Controls what should be returned when the class object is represented as a string
  - If not set, the string representation of the object is returned
  - Similar to **toString()** method in Java
  - An example WITHOUT the __str__() function:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person('John', 36)

print(p1)

Output:
<__main__.Person object at 0×15039e602100>
```

Classes and Objects

- The **__str__()** Function
  - The same example WITH the __str__() function:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f'{self.name}({self.age})'

p1 = Person('John', 36)

print(p1)

Output:
John(36)
```

Classes and Objects

- Object Methods
  - Objects can contain methods (functions that belong to the object)

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print('Hello my name is ' + self.name)

p1 = Person('John', 36)
p1.myfunc()

Output:
Hello my name is John
```

Classes and Objects

- The **self** Parameter
  - It is a reference to the current instance of the class
  - It is used to access variables that belong to the class
  - It does NOT have to be named **self** and can be called whatever
  - It MUST be the **first parameter** of any function in the class

Classes and Objects

- The **self** Parameter
  - Using 'mysillyobject' and 'abc' instead of 'self':

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print('Hello my name is ' + abc.name)

p1 = Person('John', 36)
p1.myfunc()

Output:
Hello my name is John
```

Classes and Objects

- Modify Object Properties

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print('Hello my name is ' + self.name)

p1 = Person('John', 36)

p1.age = 40

print(p1.age)

Output:
40
```

Classes and Objects

- Delete Objects and Object Properties
  - Use the **del** keyword
  - Delete an object property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print('Hello my name is ' + self.name)

p1 = Person('John', 36)

del p1.age

print(p1.age)

Output:
AttributeError: 'Person' object has no attribute 'age'
```

Classes and Objects

- Delete Objects and Object Properties
  - Delete an object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print('Hello my name is ' + self.name)

p1 = Person('John', 36)

del p1

print(p1)

Output:
NameError: 'p1' is not defined
```

Classes and Objects

- The **pass** Statement
  - Class definition CANNOT be empty
  - If we need a class with no content, put in the **pass** statement
  - This helps avoiding errors

```
class Person:
  pass
```

Table of Contents

Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class
- **Parent class (Base class)** is the class being inherited from
- **Child class (Derived class)** is the class that inherits from another class

Inheritance

- Create a Parent Class
  - Any class can be a parent class
  - The syntax is the same as when creating a normal class

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

x = Person('John', 'Doe')
x.printname()

Output:
John Doe
```

Inheritance

- Create a Child Class
  - The child class inherits the properties and methods of the parent class
  - To create a child class, send the parent class as a parameter of the child class

```
class Student(Person):
  pass
```

Inheritance

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  pass

x = Student('Mike', 'Olsen')
x.printname()

Output:
Mike Olsen
```

Inheritance

- Add the __init__() Function to Child Class
  - When added the __init__() function, the child class no longer inherits the parent's __init__() function
  - It **overrides** the inheritance of the parent's __init__() function

```
class Student(Person):
  def __init__(self, fname, lname):
    # Add properties etc.
    pass
```

Inheritance

- Add the __init__() Function to Child Class
  - To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)

x = Student('Mike', 'Olsen')
x.printname()

Output:
Mike Olsen
```

Inheritance

- Use the **super()** Function
  - Make the child class inherits all properties and methods from its parent
  - Do not have to use the name of the parent class

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)

x = Student('Mike', 'Olsen')
x.printname()

Output:
Mike Olsen
```

Inheritance

- Add Properties to Child Class
  - We can add properties to the child class, aside from the inherited properties from its parent
  - These properties are exclusive to the child class only, not the parent class

```
# Add a property called 'graduationyear' to the Student class:
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Student('Mike', 'Olsen')
print(x.graduationyear)

Output:
2019
```

Inheritance

- Add Properties to Child Class
  - The year 2019 should be a variable, and passed into the Student class when creating Student objects
  - We add the year as another parameter in the __init__() function

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student('Mike', 'Olsen', 2019)
print(x.graduationyear)

Output:
2019
```

Inheritance

- Add Methods to Child Class
  - Similar to Properties, these Methods are exclusive only to the child class, not the parent class
  - If a function with the **same name** as in the parent class is added to the child class, it **overrides** the inheritance of the parent

Inheritance

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print(f'Welcome {self.firstname} {self.lastname} to the class of {self.graduationyear}')

x = Student('Mike', 'Olsen', 2019)
x.welcome()

Output:
Welcome Mike Olsen to the class of 2019
```

Table of Contents

Polymorphism

- "Polymorphism" = "Many forms"
- It refers to methods/functions/operators with the **same name** that can be executed on many objects/classes

Polymorphism

- Function Polymorphism
  - A well-known example is the **len()** function
  - **len()** can be used on different objects

```
# STRING
myStr = 'Hello World!'

print(len(myStr))

Output: 12
```

```
# TUPLE
myTuple = ('apple', 'banana', 'cherry')

print(len(myTuple))

Output: 3
```

Polymorphism

- Function Polymorphism

```
# DICTIONARY
myDict = {
  'brand': 'Ford',
  'model': 'Mustang',
  'year': 1964
}

print(len(myDict))

Output: 3
```

Polymorphism

- Class Polymorphism
  - Polymorphism is often used in Class methods
  - We can have multiple classes with the same method name

Polymorphism

```python
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print('Drive!')

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print('Sail!')

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print('Fly!')
```

Polymorphism

```
car1 = Car('Ford', 'Mustang')       # Create a Car class
boat1 = Boat('Ibiza', 'Touring 20') # Create a Boat class
plane1 = Plane('Boeing', '747')     # Create a Plane class

for x in (car1, boat1, plane1):
  x.move()

Output:
Drive!
Sail!
Fly!
```

Polymorphism

- Inheritance Class Polymorphism
  - Child class inherits the properties and methods from the parent class
  - This means they can use the properties and execute the methods with the same name in the parent class without having to declare them

Polymorphism

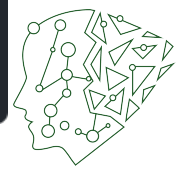
```python
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print('Move!')

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print('Sail!')

class Plane(Vehicle):
  def move(self):
    print('Fly!')
```



Polymorphism

```
car1 = Car('Ford', 'Mustang')   # Create a Car object
boat1 = Boat('Ibiza', 'Touring 20') # Create a Boat object
plane1 = Plane('Boeing', '747')   # Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()

Output:
Ford
Mustang
Move!
Ibiza
Touring 20
Sail!
Boeing
747
Fly!
```

Table of Contents

Encapsulation

- It refers to making the data private by wrapping it and its methods in a 'capsule' or unit
- This data cannot be accessed or modified outside of that unit
- Encapsulation can be achieved by making variables inside a class **private** or **protected**

Encapsulation

- Variables can be made **private** by prefixing the variable name with a double underscore __
  - Private data can NOT be accessed or modified outside the class

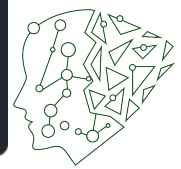
```
class Car:
    def __init__(self, brand):
        self.__brand = brand

    def move(self):
        return f'The {self.__brand} is moving now!'

car = Car('Volkswagen')

print(car.move())
print(car.__brand)

Output:
The Volkswagen is moving now!
AttributeError: 'Car' object has no attribute '__brand'
```



Encapsulation

- Variables can be made **protected** by prefixing the variable name with a single underscore _
  - Protected data can be accessed and modified outside the class
  - Adding it is simply a conventional way of informing other programmers that the variable is protected and should not be modified
  - It can, however, still be modified like normal variables

```
class Car:
    def __init__(self, brand):
        self._brand = brand

    def move(self):
        return f'The {self._brand} is moving now!'

car = Car('Volkswagen')

print(car._brand)

Output:
Volkswagen
```

Table of Contents

Abstraction

- Abstraction is about keeping the process simple by hiding unnecessary details from the user
- This reduces the complexity of the code and ensures we only concentrate on what is important
- This is achieved by creating an **interface class** (base class) and **implementation classes** (subclasses)

Abstraction

- Abstraction in Python can be done using the built-in **abc** module
  - An **abstract class** (interface class) is created by passing the **ABC** class to the parameter of the current class
  - An **abstract method** is created by applying the **decorator** **@abstractmethod** before defining the method

```
from abc import ABC, abstractmethod

class Car(ABC):
  @abstractmethod
  def car_model(self):
    pass
```

Abstraction

- The abstract class and its decorated methods are for **declaration only**, not implementation
  - This means we **CANNOT instantiate** an object of the abstract class
  - If we try to create an object, it will throw an error

```
from abc import ABC, abstractmethod

class Car(ABC):
  @abstractmethod
  def car_model(self):
    pass

car = Car()

Output:
TypeError: Can't instantiate abstract class Car with abstract
method car_model
```

Abstraction

- The implementation classes will have the same method name as that of the abstract class
  - The methods in the implementation classes will **override** the method in the abstract class

```
from abc import ABC, abstractmethod

class Car(ABC):
  @abstractmethod
  def car_model(self):
    pass

class Tesla(Car):
  def car_model(self):
    print('This is a Tesla model')

class BMW(Car):
  def car_model(self):
    print('This is a BMW model')
```

Abstraction

```
tesCar = Tesla()
tesCar.car_model()

bmwCar = BMW()
bmwCar.car_model()

Output:
This is a Tesla model
This is a BMW model
```

The End

THE END!!!